

## APPENDIX A

### Implementing a simple illumination model using POT Objects

Consider the case of implementing a very simple illumination model, such as Lambertian diffuse reflection model, using a ray tracing technique. Both the Lambert illumination model and ray tracing method are covered in 3D graphics textbooks such as *Computer Graphics, Principles and Practice* by Foley et.al., so it will not be detailed here.

The Lambert illumination model can be described by the equation

$$I = I_p k_d (N \cdot L)$$

where  $I$  is the resulting color of the point being shaded,  $I_p$  is the intensity of the light source,  $k_d$  is the material's diffuse-reflection coefficient,  $N$  is the normalized surface normal vector, and  $L$  is the normalized light vector.

Implementing this model in a ray tracer can be pseudocoded as follows:

1. For each pixels in the area to be rendered:
2. Shoot a ray from the eye point through the pixel.
3. Determine if the ray hits an object.
4. If so, determine the point of intersection  $P$ .
5. Compute the sum of the illumination at that point from all the lights in the scene. The result is the illumination value at that point.

In a recursive ray tracer, step 5 can be expanded as follows:

6. For each of the lights in the scene:
7. Shoot a ray from point  $P$  to the light.
8. Determine if the ray hits an object.

9. If not, compute the contribution of the illumination at P from the light using the equation above, using the light's intensity for  $I_p$ .
10. If the ray hits an object at point P2, the object is blocking the light. Compute the contribution of the illumination at P from the light using the equation above, using  $I_p = 0$ .

5

If all of the scene data exists in the same computer A, then steps 7-10 will be computed immediately. However, if the scene data is distributed among multiple computers, there will be a delay in sending a request to computer B, waiting for the computation to take place, and then receive the result of the computation from the remote computer. During this time, computer A

10 will not be able to proceed with the computation.

POT Engine can get around this problem by implementing steps 6-10 as follows:

- 6'. Create a shading POT Object with n input slots, where n is the number of lights in the scene. Each of the n input slots will receive the value of  $I_p$  for each of the lights in the scene.
- 7'. For each of the lights in the scene, send a raytrace request, passing the PSA of the shading POT as a parameter.
- 8'. Put the shading POT Object in a WAIT state until all the input slots receive data from the raytrace requests.
- 9'. When all the values of  $I_p$  have been placed in the input slots, compute I for each of the lights, using the equation above, and add the results together. The sum is the illumination value at point P.

Each raytrace request is implemented as follows:

25

- 10'. Shoot a ray from point P toward the light
- 11'. Determine if the ray intersects an object.
- 12'. If not, return the value of  $I_p$  for the light to an address specified by the PSA.
- 13'. If the ray hits an object at point P2, the object is blocking the light. Return zero to an address specified by the PSA.

30

The key step is step 8', where the shading POT Object is placed in a WAIT state. The POT Engine can be thought of as a large waiting queue. Every POT Object is placed in this queue when they are put in the WAIT state, and processing resources are turned over to some other process. POT Engine will constantly iterate through every POT Object in the queue, monitoring the state of the POT Objects. When all of the input slots of a POT Object are filled, the state of the POT Object will change to ACTIVE. POT Engine will take the POT Object out of the queue and allow it to execute its computation. In this case, the shading POT Object will compute the above equation for each of its input slots, add the results together, and return the sum as the illumination value at point P.

Each pixel in the area to be rendered can be computed independently of each other. Therefore, steps 1-2 can be computed in parallel. This helps ensure that the POT Engine will have enough POT Objects in parallel to keep the processing resources from being idle.

#### Extending the simple illumination model

A POT Object can be implemented using object-oriented programming languages, such as C++. By using the subclassing and virtual function features of the C++ language, the Lambert illumination model implemented above can be used as a template to implement a more complex illumination model.

As an example, consider adding specular highlights to the Lambert illumination model, by using the Phong illumination model. The Phong illumination model is also a commonly illumination model, so it will not be explained here.

The Phong illumination model can be described by the equation

$$I = I_p k_d (N \cdot L) + I_p k_s (R \cdot V)^n$$

where  $k_s$  is the material's specular-reflection coefficient,  $R$  is the normalized reflection vector,  $V$  is the normalized view direction vector, and  $n$  is material's specular-reflection exponent. All the other

variables are the same as that of the first equation. Note that the only change required to implement the new illumination model is some added computation.

A generic POT Object implementation contains all the data and functionality to perform a computation in parallel within the POT Engine. Functionality such as the ability to switch between the WAIT and ACTIVE states are implemented in a generic POT Object. For a POT Object to perform a meaningful task, programmers extend the POT Object and add any other necessary functionality via subclassing and virtual functions. For example, a shading POT Object is a subclass of a generic POT Object that implements the necessary functionality to shoot rays into the 3D space.

The Lambert illumination example is a subclass of the shading POT Object that adds the computation of the Lambert illumination equation. Therefore, a Phong illumination POT can be implemented by subclassing the Lambert illumination POT Object, adding the extra code to implement the Phong equation. A class library of different POT Objects can be created in this way, each implementing a different illumination model. They, in turn, can be used to implement new illumination models.